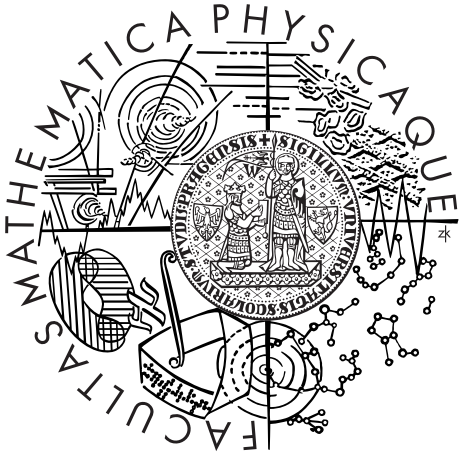
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

BAKALÁŘSKÁ PRÁCE



Jiří Harasim

Triangle mesh compression

Kabinet software a výuky informatiky

Vedoucí bakalářské práce: RNDr. Josef Pelikán
Studijní program: Informatika
Studijní obor: Obecná informatika

Praha 2012

I would like to give a special thanks to RNDr. Josef Pelikán for the great insight into a computer graphics and for the huge ammount of patience with me.

Thanks to my father for helping me and cheering me up when I needed it.

Thanks to my friends Miloš Kudělka, Martin Podloucký and Petra Cook for helping me with the correction.

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona v platném znění, zejména skutečnost, že Univerzita Karlova v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle §60 odst. 1 autorského zákona.

V ..... dne ..... Podpis autora

Název práce: Triangle mesh compression

Autor: Jiří Harasim

Katedra: Kabinet software a výuky informatiky

Vedoucí bakalářské práce: RNDr. Josef Pelikán, Kabinet software a výuky informatiky

Abstrakt: V této práci se věnuji geometrii a topologii trojúhelníkových sítí v počítačové grafice. Především se soustředím na kompresi takovýchto sítí. Navazuji na Ročníkový projekt a Softwarovou praxi, v jejihž rámci byl vytvořen program, komprimující trojúhelníkové sítě v projektu Morphome3cs, který je využíván na PŘF UK. V textu se nejprve zabývám popisem problematiky, následuje analýza dostupných dat a hledání dobrého řešení. Následuje popis samotné implementace včetně uživatelské i technické dokumentace, načež shrnuji výsledky práce, efektivitu implementovaných algoritmů a navrhuji možná rozšíření práce.

Klíčová slova: trojúhelník, síť, komprese

Title: Triangle mesh compression

Author: Jiří Harasim

Department: Department of Software and Computer Science Education

Supervisor: RNDr. Josef Pelikán, Department of Software and Computer Science Education

Abstract: This bachalor work is focused on a compression of triangle mesh problematics. It is started with analysis and trying to find a suitable solution for the Morphome3cs project. This paper is connected to the program I created as an Individual Project and Software Praxis. The program is able to compress and decompress a triangle mesh in the Morphome3cs project. There are analysed some problems and discussed a few ways to solve the problems that have arisen throughout the work. I describe implemented algorithms and present a user's and technical documentations at this paper. I close up with the possible future work, based on this solution.

Keywords: triangle, mesh, compression

Obsah

Introduction

Data compression has played an important role in computer science and information technology since the dawn of the computer era. When the hard drives were smaller and we used floppy disks to transfer data between computers, it was important to squeeze in as much information as possible. Compression was a useful tool for this. Today, we start to count capacity of hard drives in terabytes and use flash drives with several gigabytes of capacity to transport data, but compression hasn't lost its usefulness. As the amount of data we wish to transfer grows faster than the bandwidth of our phone and internet network, we need to improve our ability to compress data more effectively and faster.

This work focuses on compression of triangle meshes, which are often used in computer graphics to represent various models. More specifically, we will develop a component to a program Morphome3cs, developed on Faculty of Mathematics and Physics in cooperation with Department of Anthropology and Human Genetics on the Faculty of Science, both at Charles University in Prague. This work will offer an alternative to .obj format to save and load a triangle mesh a user works with. This format aims to occupy less space on the hard drive than .obj format.

We use the ideas and algorithms developed by Prof. Jarek Rossignac from Georgia Institute of Technology to compress connectivity and geometry of the triangle mesh. Using this as a basis, we will compress textures and normal vectors of the triangle mesh, along with the other attributes that our mesh might contain. We will also include a choice not to compress some attributes of the mesh, if the user wishes so.

1. The most useful terms

The main goal of 3D computer graphics is to display various 3-dimensional objects. We need to represent those objects in memory to be able to work with them. There are two main aproaches for the object representation:

Solid Representation

Shell / Boundary Representation

Solid models define the volume of the object they represent. They are more realistic, but more difficult to work with. They are mostly used for simulations in various fields, including medicine, engineering or architecture. One example for all solid models is CSG - constructive geometry.

Boundary models on the other hand represent only the surface of an object. They are simpler and easier to work with and are common in cinematography and computer games. Polygonal meshes, point-based representations and level-sets are the most common models here, polygonal meshes being the most used representation by far.

1.1 Triangle mesh

This work is all about triangle meshes, which are a subset of the polygonal meshes. Clearly, we are unable to represent curved surfaces with triangle meshes precisely, but we can make approximations. If we use enough triangles, the approximation is good enough for us. Ultimately, computer output devices have limited resolution as well. This means, that we don't mind triangle meshes being only approximations, because even if we had exact models, our output would be approximated by output device. Following figure *(Figure 1.1)* uses 50000 triangles and the result is pretty convincing.

Figure 1.1: 50k triangles result

The following terms are not meant to be exact, because it's not the point of this paper to build again the basics of computer graphics. They serve mainly as a reminder of known facts.

When we talk about an object we work with, we talk about a **surface model**. A surface model is a joint representation of the surface area of displayed object.

A **vertex** is a point in 3-Dimensional Euclidean space. We will work with vertices, which are unambiguously defined by their coordinates x, y and z – if there will ever be vertices u, v where $u.x = v.x$, $u.y = v.y$ and $u.z = v.z$ then inevitably $u = v$.

We define a **face** as an area on the surface model between the set of planar vertices. The **triangle** is a face which has exactly three vertices.

A **polygonal mesh** for our purpose is a set of faces connected together. A **triangle mesh** is polygonal mesh created with a set of triangles.

By a **facial normal** we mean a vector orthogonal to the face. A weighted average of all facial normals which belong to a vertex is called a **vertex normal**. Naturally, a normal in 3D Euclidean space has three coordinates.

We call a fill of any face displayed a **texture**. Every vertex has one **texture coordinate** for every face which belongs to it. We could say, that in a triangle mesh, every face has three texture coordinates, which define a texture of this face. Textures are usually 2-dimensional, because we use them as a fill of faces – already 2-dimensional objects.

A **2-manifold** is a topological space whose points all have open disks as neighborhoods.

A **handle** is simply said a torus incorporated into the mesh.

If we identify a **hole** in the mesh, we found a place where we could fit in few faces and dont break a 2-manifoldness.

1.2 Compression

We define a few useful terms in this section for compression in general and we present a corner table data structure used in the Edgebreaker algorithm. We will see that a corner table is very efficient data structure to represent a triangle mesh.

1.2.1 Compression in general

Compression itself is divided into two main areas:

lossless compression

lossy compression

Lossy compression is used mainly to compress music, films and graphics files, photos and various complicated images. It serves the need of having smaller files to fit some smaller devices, such as IPod, cell phone and other devices. It is based on the fact that we don't need the best possible quality for these devices, because they are not capable of displaying them or playing them at full quality anyway. So we sacrifice some details and we preserve only the main themes of the file or we create approximations. There are some well known lossy compression formats. Here are just a few of them:

MP3 or WMA for music files

AVI, MPG or WMV for video files

JPG as most common image lossy format

Lossless compression serves to preserve data and programs. Approximations are out of the table, as well as sacrifice of any data - nobody would want a disertation paper with some letters missing for example. We also use lossless compression to compress pictures which are not that complicated. These lossless compression data formats are generally known:

PNG or GIF for pictures

ZIP, RAR or TAR for data programs

The original Edgebreaker algorithm, presented by prof. Rossignac, was lossy algorithm. He used big models (a car for example) and rounded the decimal places of model's vertices coordinates to a acceptable degree. Our objectives in

this paper are slightly different than that. We don't need to present really huge efficiency for our algorithm like prof. Rossignac did, because Edgebreaker already proved for its effectiveness. As we work with hundreds of thousands of vertices and triangles on a very small area (human faces), we need to preserve the precision in the first place – our version of Edgebreaker algorithm will be lossless. It will result into compression not being as effective as it was with the lossy Edgebreaker, but still effective. We will add a compression of other attributes of a triangle mesh to the algorithm as well.

We now define some terms which will be used at the end of the paper to confront the effectiveness of our compression compared to uncompressed format .obj and to the original Edgebreaker data.

Original data requires a certain ammount of space on the hard drive. We mark the bits needed to save the original data as α and the bits needed to save the compressed data as ω. We define the **compress ratio** as $\lambda = \frac{\omega}{\alpha}$.

We call the time needed to compress data the **compression time**. We will measure the time needed to compress triangle mesh using different parts of our algorithm.

After the Edgebreaker compression, the work is not done yet. For example the clers string is often about half a million letters long and contains the letters C, L, E, R and S only. The Edgebreaker algorithm aims to create data patterns in its output, which leads to the possibility of compression of this data with an Entropy coder.

An **entropy coder** is the lossless compression algorithm, which searches for patterns in raw bits and compresses them. The groups of bits with the highest probability use the least space in the output and vice versa.

1.2.2 Corner table

We now come to a description of a corner table data structure. The principles described in this section are directly implemented into the compression scheme in Morphome3cs. The Edgebreaker algorithm uses corner table specifically as its only data structure. It recieves corner table and produces compressed data, or it recieves compressed data and restores the corner table from them.

Every face in the triangle mesh has three **corners**. They corelate with vertices to which is the face bound. Corner knows the triangle it belongs to and its vertex as well.

Let us have a corner c and triangle t which it belongs to. We define the **opposite corner** to the corner c as a corner d. d is a corner in the adjecent triangle, where both c and d are unique vertices for these two triangles.

The corner table consists of two integer arrays – O and V – with three times as many integers as there are triangles and from a list of vertex data. Every triangle is represented by its three corners. They are stored consecutively in these two integer arrays. This system provides us with free corner – triangle relation. When we have a corner, the integer division of its order in array provides us with the triangle number. Likewise, when we have a triangle number, we can simply find its corners by multiplying triangle number by three and adding zero, one and two. The corner order in a triangle is such, that it respects clockwise or counter-clockwise orientation in the whole mesh. That allows us to iterate through one triangle's corners by calling the next or previous function.

The V table value refers to a list of vertex data, associated with this particular corner. It contains vertex coordinates in the first place, but we can refer to any vertex data we add to our vertices with this system.

The O table contains index to the V table, to the opposite corner. This structure allows us to simply refer to every corner in given triangle as well as to all the adjecent triangles as shown in the Figure 2.1. Because we have an orientation defined throughout the whole mesh, we can define which adjecent triangle is left and which is right with respect to triangle, we came from using the opposite corner.

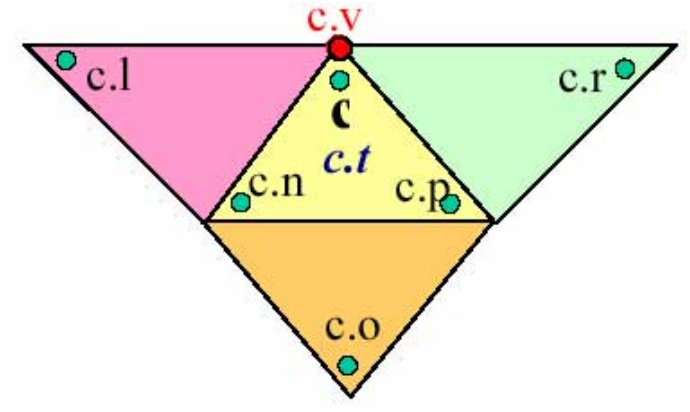


Figure 1.2: A corner table data structure on adjacent triangles

2. Problem formulation

2.1 Morphome3cs

As stated in the Introduction, Morphome3cs is developed on Faculty of Mathematics and Physics for Department of Anthropology and Human Genetics on Faculty of Science. Morphome3cs is program running on Windows platform developed mostly in C#, although it uses Python scripts, R calculations and Open GL libraries as well. It was developed by two teams as a software project. Lectures did various work on it and students contributed in a form of their Bachelor projects as well.

The project's goal was to create stable independent open source software, which provides basic morphometric operations mostly on human faces. It is relatively easy to use, because we don't expect scientists to do the programmers work. It provides a various operations used for research in anthropology, genetics or other areas.

To be able to develop software in Morphome3cs framework, we must have installed MS Visual studio 2008 with .NET version 3.5 or higher. For coordination, archiving and versing, project uses Apache Subversion – SVN. Everything else we need for installation of this software, such as python, R and other programs is stored in its SVN repository on the address svn://cgg.mff.cuni.cz/morpho/trunk. The more detailed manual is part of a repository itself.

Our work is only a small part of the project. It provides alternative to .obj files to save meshes. It is a quite smaller alternative, but slower with saving and loading. It is used in the 3D viewer part of the program. We use our own suffix to the files – EBC.

Morphome3cs works mainly as a series of connected filters called workflow, where output of one is often input of another. This solution allows filters to be easily replaced with alternatives to provide variety of functions for user.

The important elements of Morphome3cs are the following:

Morpho main is a basic class of the whole program. We call everything else from here, or hierarchically from here.

Data structures are used in various classes. The data structure Mesh is important for us. Generally we have slightly different needs for our data structures in Compression than in the other parts of the project, so we will define few data structures for our own needs.

GUI provides user forms for some filters and parts of the program.

Filters class, intuitively, provides implementation of many filters.

Specimen editor is the starting point for majority of users. It is possible to run workflow to a collection of specimen user has.

Mesh mending is able to clear a triangle mesh from some artefacts which may occur during the scanning of human faces. This is important for us, because it has a method to fill holes in the mesh for us.

Python scripts represents workflows.

Viewer 3D renders a 3D scene. Meshes shown here can be saved in our defined .EBC format.

Viewer 2D renders a 2D scene.

InOut3DSupport provides the saving and loading of 3D meshes. It does play only a subtle role in the Morpho project, but it provides interface to save and load meshes, which is used by our Compression. This is very important for our project, as the compression is controlled by a few methods in this class.

The **Compression** is our work here. It implements the Edgebreaker algorithm and provides support for it. Details will be discussed later.

2.2 Mesh in Morphome3cs

The Mesh structure in Morpho represents a triangle mesh. It is declared in Data structures section of a code. It stores everything we need to know about the triangle mesh we work with. When the Viewer 3D renders a face, it uses this mesh. Hence we need to be able to convert a mesh to a corner table and a corner table back to a mesh. To do this, we need to examine the mesh structure closer and find out how does it work and what is it composed of.

The mesh structure consists of vertex data, texture data, normal data, faces and additional vertex information. Vertex and normal data are represented by thier three coordinates. Texture coordinates are stored as two texture coordinates and a zero as a third coordinate. All of these data arrays have defined certain order. The face representation refers to this defined order. Each face consists of three integers which refer to the vertex data. If the mesh has textures and normals, they are stored for every face as well using similar principle. Very useful piece of information here is that the normal data are stored for faces, but are in fact the vertex normals so they can be looked at as a vertex property. Additional information are not used right now in the project, and because quite anything can be attached to vertices, we don't compress additional vertex information. It doesn't mean we will not save it if it appears, it only means that we will store it in its raw form.

2.3 Problem summary

We need to implement a functional version of the Edgebreaker algorithm as a core of our project.

To save a mesh, we need to construct an algorithm which converts it from Morphome3cs Mesh to a corner table. Then we run our implementation of the Edgebreaker algorithm, extended by our procedures to compress textures and normals, and save the result into our new format EBC.

To load a mesh, we will take the file in EBC format and execute the decompression part of the Edgebreaker algorithm along with decompression of textures and normals. When this is done, we will convert obtained corner table back to the mesh format in Morphome3cs.

This process is lossless compression, so we must obtain the same triangle mesh we have started with. To verify the result, we will implement an algorithm to check whether two meshes match each other or not.

3. Solution

We start this section with presenting the Edgebreaker algorithm. We follow with closer look at the data we work with in general and with few ideas about what might work and what we want to test. After that, we will present algorithms which are actually implemented into the project to obtain desired data structures and to compress properties of the mesh which the Edgebreaker doesn't compress by itself.

Every vertex in the triangle mesh has its three coordinates and every triangle has its three vertices. We call vertex coordinates the geometry of a triangle mesh and the triangle incidency, common edges, common vertices etc. the connectivity of a triangle mesh. The compression of connectivity and geometry is well handled by the Edgebreaker algorithm.

3.1 The Edgebreaker

The Edgebreaker algorithm works in steps. It removes one triangle from the actual mesh in every step it performs. The actual mesh is composed of one or more regions, which all can be iterated through using the gates. The gate is simply an edge in the triangle, which lies between opposite corners. The gate always lies at the border of the actual mesh. It is called a gate, because it is the gate between already compressed and not compressed parts of the triangle mesh. One step can be described as a series of actions. Note that all the pictures in this section were adapted from the Edgebreaker papers published by prof. Rossignac.

3.1.1 Compression part

We obtained a gate at the end of the last step, or as an input. We use this gate to proceed into a triangle adjacent to it which we haven't visited before. Based upon what is the current situation around this triangle, we write one of the symbols C, L, E, R or S into the clers string. We might write down vertex information we want, we might write down the face information we want also and we choose another gate and continue. The descriptions of individual situations follow:

- When we encounter a vertex we haven't visited before, we write the **C symbol**. This is the most common situation, as it occurs at approximately half of the cases. It is the only situation in which we are adding new vertices into the compressed mesh. In every other situation, we describe faces only. When we encounter this situation, the Edgebreaker algorithm encodes this vertex coordinates by using a paralelogram prediction scheme.

In our implementation in Morphome3cs, we encode vertex normal, default texture coordinates for vertex and other vertex properties along with the original coordinates.

- We write the **R symbol** if we encounter following situation: The vertex, that the gate leads us to, has been visited already and it is on the right side of the gate. This situation is very common too, due to the Edgebreaker algorithm preference in iterating through the mesh.
- The **L symbol** is similar situation as the R symbol is. The only difference being that we have already been on the left side of the gate instead of the right side of the gate.
- We encounter the **S symbol** when we visit a vertex, which has been visited already, but neither of the left or right triangle adjacent to the gate has been visited. In this situation, we call the compression scheme recursively for the right triangle and when that procedure finishes, we continue to the left. It is possible for us to return back to this particular triangle, because we work with handles. If this situation happens, the active procedure ends itself, because we have explored and compressed everything in this part of the mesh and we mark the handle.
- Finally, we use the **E symbol** when the vertex has been visited already and both left and right triangles have been visited as well. This symbol ends current compress procedure, because we have compressed everything we could here.

These situations are shown on the following figure. The entrance gate is always located at the very bottom of every diagram.

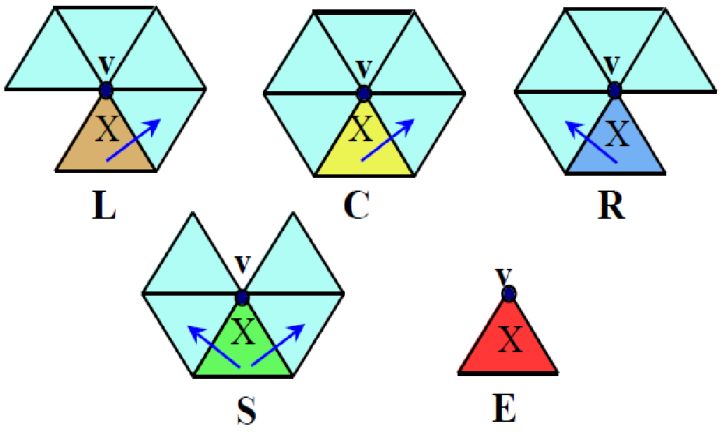


Figure 3.1: CLERS situations

An example of the work of the Edgebreaker algorithm. The compression just started at the green triangle and as we can see, it continues determinable using a very simple pattern.

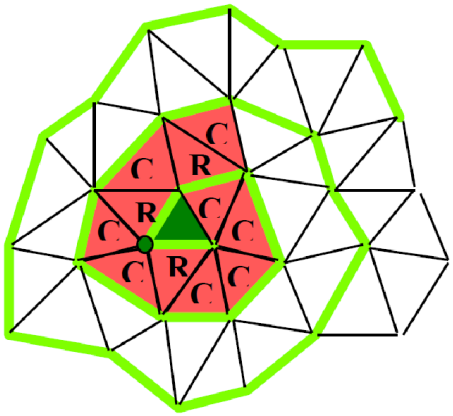


Figure 3.2: Beginning of the compression example

And the last example. This is the final stage of a compression, where the surrounding mesh has been compressed already. The green arrow is pointing at the entrance gate here.

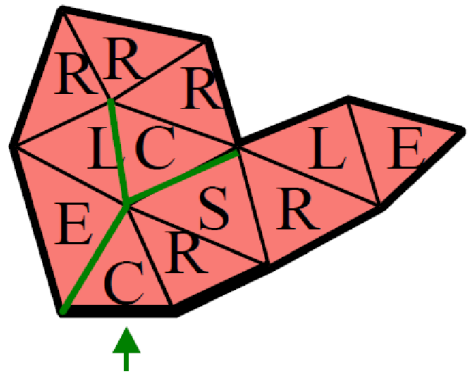


Figure 3.3: Final stages of the compression example

We have been talking about the compression part of the Edgebreaker algorithm so far, but that is only one part of it. So in following lines we will explore the decompression part, how it works and how it restores the triangle mesh from the saved data.

3.1.2 Decompression part

The decompression part takes the clers symbols one by one and constructs the corner table by appending new triangles to the previously visited triangles. We again describe situations, as we did with the compression scheme:

– If the symbol is a **C**, the algorithm stores the -1 as the corner opposite to the left edge. This temporary mark will be later overwritten by the zip

method, and will be replaced by a proper opposite corner number.

- When the symbol is a **L**, the algorithm assignes -2 with the opposite corner and tries to zip with the edge on the left.
- When the algorithm encounters a **R** symbol, it assignes -2 value to an opposite corner and the zip method is not called.
- When the symbol is a **S**, the algorithm calls recursively the decompression method and zips the part of the mesh, which is incident with the right edge of active triangle. After it is done, decompression continues at the left side.
- Finally, when the symbol is an **E**, algorithm assignes -2 values to the both adjacent corners and it calls the zip method iteratively. The zipping continues as long as there is a free edge with -2 value on the one side and free edge with -1 on the other side.

The concept of the connectivity decompression is pretty much described by this, the only thing remaining to explain is how does the zip method work. This method is given one corner with -2 mark. It looks for -1 mark by iterating through mesh backwards, by going through left edges of the left triangles. When it succesffully finds -1 value, it connects the corner as opposite corner to the one with -2 value. It then returns back by iterating through right edges of right triangles, looking for a -2 again. If it finds another -2 value, it goes back, looking for -1 again and so on. It stops, when it comes back to the starting triangle.

Following picture gives us a little bit of insight into this problem:

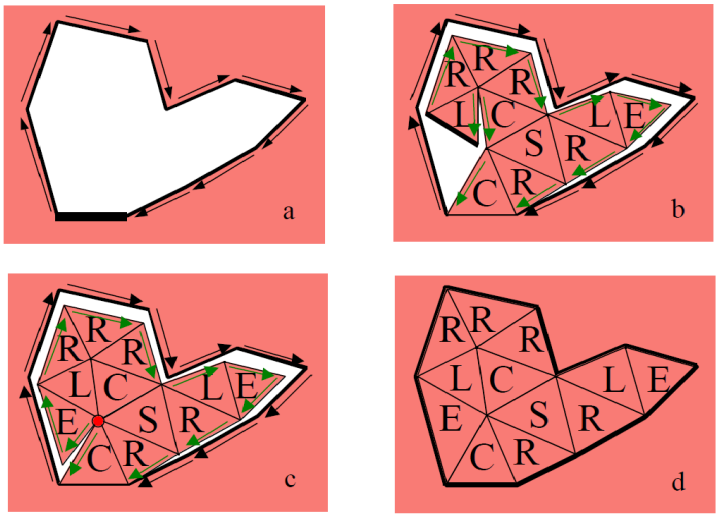


Figure 3.4: Example of zipping the last part of the mesh

We have decompressed the connectivity part of the triangle mesh right now, which means we know how are the triangles connected with each other, which vertices are similar for which triangles, we know the common edges of the adjencent triangles etc. What we don't know yet, is the geometry part of the triangle

mesh. So we go through the mesh in the same order once again and we count the vertex coordinates for every C symbol we encounter. We encoded the vertex data using the paralelogram prediction scheme and we decode them with it as well. We have the core of the triangle mesh restored from the input file right now.

3.2 Analysis

The set of triangle meshes we work with is well specified. It is a human face, which typically consists of around 200000 vertices and around 400000 triangles. Exact numbers fluctuates a bit, but it is not that important. Obviously, these are huge numbers to work with. We assume 2-manifold equivalent connected meshes with no holes. Handles might be present in the mesh, our algorithm can handle them. Morphome3cs contains a tool, which can be used to modify meshes which doesn't meet this criteria. The mesh mending tool can be used for filling holes, repairing overlaying faces, removing zero volume faces and other actions in order to compress the desired triangle mesh.

3.2.1 Compression of vertex normals

The vertex normals contain three coordinates , just like the vertices do. The Edgebreaker compression algorithm works by iterating through the whole triangle mesh, encoding connectivity and geometry along the way. That means that any vertex property can be encoded along with the geometry data. We will use this feature to compress vertex normals. When I was exploring various possibilities of compression for vertex normals, this wasn't the only solution that occured to me. Another possibility was to use the same trick which is used in order to create obj file from the mesh – to look at vertex normals as a face property. That would allow me to compress vertex normals along with the connectivity of a triangle mesh, instead of a geometry. I explored this possibility and I have found compression along with geometry data vastly superior.

The reason for the superiority of the vertex property approach is at hand. There are approximately 200000 vertices and 400000 triangles in the triangle mesh. If we use the connectivity approach, we have to compress 1200000 pieces of information (three for every triangle). Although most of it are bits saying that we don't have anything here (it is enough to store the information for every vertex only once and every vertex has multiple faces), we still need to compress a normal for every vertex. In another words, by using this approach we will compress the same amount of data as with the vertex property approach plus something additional which has no use. Plus it will be much harder to create and work with any predictions by the face property approach.

I implemented both algorithms into the solution merely out of curiousness only to find out, that the vertex property approach using paralelogram prediction in

fact needs approximately three times less amount of space than face property approach on multiple meshes. Naturally, I stepped away from the face property approach. The compression module in the Morphome3cs project therefore uses the vertex property approach to compress the normals.

3.2.2 Compression of texture coordinates

The texture compression is a little bit more complicated than the compression of normals. It is caused by few factors. We will do simple math now. We have around 400000 triangles, every one of them has three vertices. That means, we need to compress close to 1200000 texture vertices. Although texture vertices have three coordinates in Morphome3cs, we know for granted that the 'z' coordinate is zero in every circumstance. So we have two coordinates to compress for every texture vertex. This makes for around 2400000 texture vertices coordinates to compress.

There are few possibilities again we wish to explore here. When I was thinking about texture compression, it occured to me that I want to know some statistics about how exactly texture vertices are related to mesh vertices. I ran a few tests and the results were very surprising. In triangle meshes Morphome3cs uses the most, 99% of texture vertices have only one real mesh vertex which they are connected to. In another words, when we get a texture vertex, there is a 99% chance that in every occurence of this texture vertex, it will always have the same mesh vertex which the texture vertex is assigned to. The other way around is interesting as well. 95% of mesh vertices have only the one texture coordinate assigned to it.

Because every face has its texture, we can encode every texture at the time we encode the clers string character. Or we could call it encoding along with the connectivity data. This will provide us with efficient order of compression and decompression throughout the triangle mesh. Now the first possibility we can think of is the simple way of encoding texture coordinates in sorted order and refer to its order number to tell which texture coordinate belongs where. We will use the sorted order of texture coordinates to encode them with a little bit of prediction, because we can count the average difference between them. As we don't use anything correlated to the iteration through the mesh nor we use any other space saving technique, we don't expect this possibility to be very effective, but it is the first step we can think of.

Another possibility is to use the triangle mesh iteration process to estimate actual coordinates and store the difference between our estimate and the reality. We will not use the paralelogram prediction scheme here, because with texture coordinates we simply don't expect the opposite corner texture coordinates to have much in common with our new texture coordinates. A 95% of mesh vertices has its only texture partner. The only one. When we visit a triangle, we already have visited and stored texture coordinates for two of its corners and this coordi-

nates have at least 95% chance of being correct, each! The real chance is slightly different, because some vertices have more than one set of texture coordinates assigned to them. On the other hand, this principle aplies for each of the two corners, so we multiply the odds together.

So what we do is quite simple. When we visit a new triangle, we assume that every texture coordinate used previously with its vertex is the right one and we store the difference between our assumption and the reality. If we check the vertex with multiple texture coordinates assigned to it, we will use the last one stored. We can visit triangles with five different clers symbols. With L, E, R and S we know that every vertex was visited before, so we predict texture coordinates for every corner and continue. With C, the situation is different, but not that much. We have two texture coordinates which are the same as the two of previously visited triangle. We will predict the last coordinate in the middle of these two. Written on the paper, this approach looks quite promising.

Last option I thought of was slightly different. The most important idea here is to look at the texture coordinates as the vertex property rather than the face property. This is allowed by statistical numbers we acquired – 99% of texture coordinates has its one and only vertex and 95% of vertices has its one and only texture coordinates. We will create one to one relation between vertices and texture coordinates and store additional (or we could say not used) texture coordinates separately. When we iterate through the triangles, we will say if we use the default texture coordinates for the particular vertex or we refer to another texture coordinates stored elsewhere. If we create a good vertex-texture relation, we don't expect a whole lot of texture coordinates to be stored separately.

99% looks higher and better to work with, so we might try to create relation based on texture coordinates – vertex pair. After a closer look, we have encountered a few problems though. What if two or more texture coordinates would be related to the same vertex? What do we do with coordinates which just don't have a single vertex assigned to them? Are we sure to have every single coordinate covered up? More questions have arisen here. We could store every problematic texture coordinate in separate file to fix this, plus we would need to check upon every vertex to have all of its texture coordinates covered. It looks complicated, but possible.

What about the other way around? A 95% is slightly less then 99%, but it still looks good enough. Additionally, we store the vertex property instead of the face property, and we have less vertices than we have faces. To create vertex – texture coordinates relation seems only natural here. So assume we do this. Now we have relation for our 95% of vertices done by relating them to its only texture. What to do with the rest? I came up with two possibilities here. I can simply store the most common texture coordinates for every vertex as its relation. Other coordinates would be stored separately. It would mean, that some textures would be stored in both ways – as a relation and separately. On the other hand, it would mean that the default texture coordinates in the iteration through mesh phase would appear as often as possible.

The other possibility would be to create a complex system which tries to find an ideal solution and store exactly those texture coordinates that would result in the best compress ratio possible. After a brief look into this approach I stepped away from it, because it would be really complicated and the other way seems good enough in itself.

As I tried both, the vertex – texture coordinates relation and the texture coordinates – vertex relation on multiple meshes, I have chosen the vertex – texture coordinates relation, because the results of compression were almost the same for both of them and this one was far easier to create and to work with.

After implementation of these three options, I was really surprised. The first one, the one wich sorts the coordinates only, was not surprising at all. It was only a bit better than the raw-stored-data. Nothing to worry about, it never had an ambition to be effective. It was the second option that was really disappointing for me, the one that tries to estimate the right texture coordinates. It was better than the first one, but was far worse than I have expected it to be. It saved only about 10% of space in comparison with the raw-data. The third option turned out to be superior by far, as in avarage it compresses texture coordinates to around 40% of the original ammount.

It is quite simple with the decompression of both, textures and normals. The Edgebreaker decompression provides us with the iteration throughout the whole mesh and since we used the Edgebreaker compression iteration to encode desired data, we know that if we decode data in this order, we will have what we started with for every vertex and face. We can be sure that the iteration order is right, because the Edgebreaker algorithm works.

3.2.3 Other methods

We will think about the algorithm which checks whether two meshes are the same and about a converter from a mesh to a corner table and back.

Why is algorithm for checking two meshes needed at all? Why can't we just compare them?
The answer for both these questions is at hand. We obtain a mesh from outer source. We compress it by iterating through it and then we decompress it. The result will be a mesh with vertices which order is defined by the iteration through the mesh. Unless the vertices and faces in the original triangle mesh were ordered in exactly this iteration order, their positions in the representation will change. It doesn't mean the mesh is different, we have changed the labels of vertices and faces only. Every important piece of information is at the right place – or at least should be, if we did our work properly. Thus we design an algorithm to check upon these two meshes and inform us if they match or not. This algorithm will not be used in the compression scheme used in Morphome3cs, but is integral part of our work.

There are few things we need to check to be assured that two meshes are exactly the same. First set of controls will check upon the coordinates of vertices, textures and normals. We will order the coordinates by x, y and z values and compare them against each other. Next thing we need to check is, if the triangles, textures and normals use the same values in both meshes. To check normals, we simply convert them back to the vertex property and we compare the values through the vertex sorted order. To check if the faces and textures are the same, we need to sort the faces.

When we sorted vertices, their order has changed and they have a new defined order now. We must update faces so they respect this new order. We do it by simply remembering the original position of a vertex and remembering where it moved by sorting. As it doesn't matter which vertex in the triangle is the first, which is the second and which is the third, we can sort faces inside them to be arranged such as the first vertex has the lowest index number and the last vertex has the highest index number. As we move vertex indexes inside the faces, we must not forget to move the texture coordinates as well. We can now sort faces by the first, the second and the third vertex index numbers. Both meshes should now have the exactly same list of faces and texture coordinates. If both meshes fulfil every check we made, we can declare that both these triangle meshes are exactly the same.

Last thing open for analysis is a way to convert a mesh to a corner table and back. A mesh has every bit of data we need to create a corner table, with one exception. A mesh doesn't recognize neighbour triangles, which implies that we don't know our opposite corners in the corner table. It means, that for every pair of vertices of a face, we need to find exactly the same pair present in another face. The assumption of mesh with no holes is the key here, since if we have some holes in the mesh, we are not sure if such a pair exists. To do that, we again sort face vertices so the smallest number is the first and the highest number is the last.

We create two arrays of faces now. In the first array, we sort the faces according to the first vertex number, the second vertex number and the third vertex number. We remember the index in this array for every face. In the second array, we sort faces according to the second vertex number, the third vertex number and the first vertex number. We will mark every unasigned corner and unvisited face. We create a queue of triangles and process it.

One step of queue processing follows: we de-queue an unvisited triangle from the queue and we find a pair for every two vertices in its face by the halving method in our two sorted arrays. We enqueue every unvisited triangle our current triangle is adjacent to. This algorithm clearly iterates through the whole mesh, if the mesh meets the criteria we assumed it does. The assumption of connected mesh is the key here. We can be sure that the opposite corners in the corner table are connected to each other right now.

Convert a corner table to a mesh is quite easy, because it already contains everything mesh does. So we just create proper data structures for a mesh and

transfer the data into them. Since we treat normals in the corner table like vertex normals, but they are stored in the Mesh in Morphome3cs as facial normals, we can convert the vertex normals to the facial normals simply. We will assign nothing to every corner to begin with. We choose one corner at each vertex. This corner will contain vertex normal and every other corner will not change. The result is obviously what we wanted – we created the facial normals representation for vertex normals. That would be all for converting a corner table back to a mesh.

3.3 Algorithms

We present the pseudocodes of various algorithms implemented in the Compression module of Morphome3cs project. We start with the algorithms for compression of textures and normals. Then we will show the algorithm used for converting mesh into a corner table and back. We close up with a checking algorithm, which determines whether the starting mesh and the result are the same.

3.3.1 Normals and vertex property

What do we present here is not quite an algorithm, because we don't need any algorithm to compress vertex normals. We simply create a new array of whatever we wish to compress for every vertex and then we make sure that the vertex index in vertex array correlates with index of its property in this new array. This is useful for our texture compression as well, since we treat some of the texture coordinates as a vertex property.

The only thing that needs a little bit of thinking about is converting the facial normals into the vertex normals and back. It is pretty straightforward anyway, so only a little bit of thinking is needed.

The face in the Mesh data structure in Morphome3cs contains both, the facial normal identificator and the vertex identificator. So we will iterate through all faces in the mesh, pairing up vertex index with facial normal index. As we know that it is in fact the vertex normal itself only stored this way for some inner reasons in Morphome3cs, we do not need to do anything else than create a new array of vertex property and store the normals into the array with an index, that is the same as the vertex index which it is paired up with.

Converting the vertex property back into the facial property is even simpler. We will just store the acquired normals into the normal list in the Mesh data structure where the normal index in every face will be the same as the vertex index in the same face or index of empty normal if the vertex had normal assigned to it already. We need to create one indexing array to mark already used vertices, to avoid unnecessary duplicites.

In the Edgebreaker algorithm compression scheme, whenever we save the C symbol, we encode any vertex property along with the vertex data. For normals, we use the paralelogram prediction scheme and for the texture coordinates property we just store the raw data.

In the decompression scheme, whenever we read the C symbol, we decode this property along with vertex data. We decode normals using the paralelogram scheme again and we just read the data from the file for the texture coordinates.

This method of storing the vertex data works obviously correctly, because the Edgebreaker algorithm itself works correctly.

3.3.2 Texture compression

Algorithm presented here is used for determination of the vertex – texture coordinates relation explained in the analysis section. Texture coordinates which any vertex uses and are not assigned to this particular vertex as default value are stored separately and handed to the compression algorithm along with the relation array.

```
Data: texture coordinates, number of vertices, face data
Result: Vertex-texture coordinates relation table, other texture
        coordinates table
initialization;
VTrel = array of ints ;
A = set of ints;
S = array of set of ints with as many elements as there are vertices;
L = array of list of ints with as many elements as there are vertices;
foreach face do
    foreach vertex index do
        S[vertex index].add(texture index);
        L[vertex index].add(texture index);
    end
end
foreach vertex do
    if S[vertex].count is 1 then
        VTrel[vertex] = S[vertex].At(0);
    else
        VTrel[vertex] = S[vertex].At(i) i: maxcount from L[vertex];
        foreach S[vertex].At(i) i: not maxcount from L[vertex] do
            A.add(S[vertex].At(i));
        end
    end
end
```

Algorithm 1: An algorithm to create a vertex-texture coordinates relation

The compression algorithm receives the relation table and the additional data table. The relation table is encoded along with vertex property, because it in fact is a vertex property too in this approach. The additional data table is stored at the start of the compression, because it doesn't quite use anything related to the Edgebreaker compression of the mesh.

Whenever the Edgebreaker algorithm writes into the clers string, that is the right time to compress the texture for the visited face. We have separate file for this, something like clers but used for marking textures. We will first store the

corner we came to this triangle. We check if the texture is the default texture for this vertex. If yes, we write 'D' into the marking file. If it is not, we write the index of these texture coordinates from the additional data table. When this is done, we do exactly the same thing for previous corner and next corner in this triangle in this order. The order of these actions is important when decompressing.

The decompression part of this is just going backwards. Whenever we decode any symbol from the clers string, we also decode three symbols or numbers from marking file. Because the connectivity of the triangle mesh is decoded sooner than geometry, we need to store these symbols and numbers as a corner property for now. After we decode the vertex property and additional data, we can store all texture coordinates in an array. As our algorithm presented above might have created some duplicities, we need to iterate through all texture coordinates and make them unique again. One easy way to do this is just insert them into the HashSet data structure presented in C#language and remember the index or D's used previously. We replace all D's and index number to the additional data table by indexes to this array.

3.3.3 Converting algorithm

As stated before in the analysis section, there is nothing special to mention about the conversion from the corner table to the Mesh. It gets much more interesting with conversion from the Mesh to the corner table though. We described the principle of the conversion back in the analysis. We will now look closer on the algorithm described back there.

Data: mesh – vertex coodtinates and faces
Result: corner table
initialization;
O = table of opposite corners V = table of vertex data for corners
rearange each face to have the first vertex with the lowest index and the last with highest index **foreach** *opposite corner* **do**
  set value to -1
end
S1 = sort faces with respect to the first, the second and the third coordinate **foreach** *face* **do**
  store its current index assign values to the V table – every faces bounding vertices to three consecutive elements in array
end
S2 = sort faces with respect to the second, the third and the first coordinate
Enqueue the first triangle and mark the first triangle as visited **while** *queue is not empty* **do**
  remove triangle from a queue find the same edge for all three edges in a triangle by halving the interval in S1 or S2 **if** *the found triangle wasn't visited yet* **then**
    Enqueue triangle Check orientation of the vertices and if it is not right, repair it
  end
  Pair up opposite corners
end

Algorithm 2: An algorithm to convert a mesh into a corner table

There is one particular thing that has not been mentioned before. We want our mesh to be oriented. It doesn't matter if clockwise or counterclockwise, simply oriented. It means that every triangle has its vertices in the order that respects this orientation. When we process the first triangle, we respect its original orientation and orient every following triangle accordingly.

3.3.4 Checking algorithm

The following algorithm was described in detail in the analysis section. The pseudocode follows.

Data: mesh M1, mesh M2
Result: are the meshes the same?
initialization;
sort vertex, texture and normal coordinates in both M1 and M2 reindex all faces to respect this new order in both M1 and M2 **foreach** *vertex, texture and normal from M1* **do**
    check if it is identical with the vertex, texture and normal with same index in M2
end
rearange faces so the first vertex index is the lowest and the last the highest in both M1 and M2 sort rearanged faces with respect to the vertex indexes in both M1 and M2 **foreach** *face in M1* **do**
    check if it has the same vertex, normal and texture indexes as the face with the same index in M2
end
Algorithm 3: An algorithm to check whether two meshes are identical

The checking algorithm is used for internal purposes only. It is never called in the normal execution of the Morphome3cs project.

3.3.5 Entropic coding

After the compression with our compression module, we obtain data which can be compressed more by entropy coder. Because it is not the point of this paper to go into entropy coding, we simply adapt an entropy coder with suitable licence found on the internet. This coder was implemented by Mr. Chris Lomont and it is a simple arithmetic coder. This coder is incorporated right into the compression module. Some parts of the data compressed by the Edgebreaker algorithm as well as our extensions of it are designed to be compressed more by some kind of arithmetic coder. The clers string or the texture marking file contains only a few symbols and the paralelograms create a lot of data which looks like 0.00.... Arithmetic coder is built to compress data with repeating patterns quite effectively.

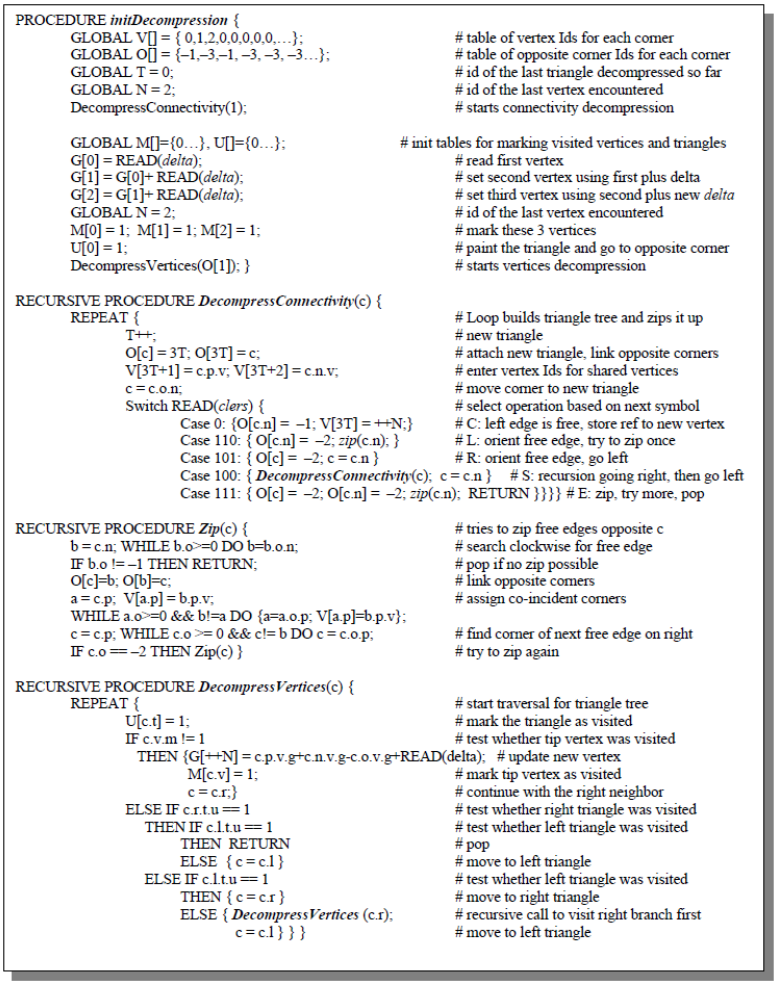
3.3.6 The Edgebreaker

We present the Edgebreaker algorithm pseudocode, because it just should be here. It is directly adapted from the original Edgebreaker paper.

```
PROCEDURE initCompression (c){
        GLOBAL M[]={0…}, U[]={0…};          # init tables for marking visited vertices and triangles
        WRITE(delta, c.p.v.g);              # store first vertex as a point
        WRITE(delta, c.v.g – c.p.v.g);      # store second vertex as a difference vector with first
        WRITE(delta, c.n.v.g – c.v.g);      # store third vertex as a difference vector with second
        M[c.v] = 1;  M[c.n.v] = 1; M[c.p.v] = 1;   # mark these 3 vertices
        U[c.t] = 1;                         # paint the triangle and go to opposite corner
        Compress (c.o); }                   # start the compression process

RECURSIVE PROCEDURE Compress (c) {                                  # compressed simple t-meshes
        REPEAT {                                                    # start traversal for triangle tree
                U[c.t] = 1;                                         # mark the triangle as visited
                IF c.v.m != 1                                       # test whether tip vertex was visited
                 THEN {WRITE(delta, c.v.g – c.p.v.g – c.n.v.g + c.o.v.g);  # append correction for c.v
                        WRITE(clers, 0);                            # append encoding of C to clers
                        M[c.v] = 1;                                 # mark tip vertex as visited
                        c = c.r}                                    # continue with the right neighbor
                ELSE IF c.r.t.u == 1                                # test whether right triangle was visited
                   THEN IF c.l.t.u == 1                             # test whether left triangle was visited
                        THEN {WRITE(clers, 111); RETURN }           # append code for E and pop
                        ELSE  {WRITE(clers, 101); c = c.l }         # append code for R, move to left triangle
                   ELSE IF c.l.t.u == 1                             # test whether left triangle was visited
                        THEN {WRITE(clers, 110); c = c.r }          # append code for L, move to right triangle
                        ELSE {WRITE(clers, 100);                    # append code for S
                                Compress(c.r);                      # recursive call to visit right branch first
                                c = c.l } } }                       # move to left triangle
```

Figure 3.5: The Edgebreaker compression algorithm pseudocode



```
PROCEDURE initDecompression {
        GLOBAL V[] = { 0,1,2,0,0,0,0,0,…};                  # table of vertex Ids for each corner
        GLOBAL O[] = {–1,–3,–1, –3, –3, –3…};               # table of opposite corner Ids for each corner
        GLOBAL T = 0;                                       # id of the last triangle decompressed so far
        GLOBAL N = 2;                                       # id of the last vertex encountered
        DecompressConnectivity(1);                          # starts connectivity decompression

        GLOBAL M[]={0…}, U[]={0…};                  # init tables for marking visited vertices and triangles
        G[0] = READ(delta);                                 # read first vertex
        G[1] = G[0]+ READ(delta);                           # set second vertex using first plus delta
        G[2] = G[1]+ READ(delta);                           # set third vertex using second plus new delta
        GLOBAL N = 2;                                       # id of the last vertex encountered
        M[0] = 1;  M[1] = 1; M[2] = 1;                      # mark these 3 vertices
        U[0] = 1;                                           # paint the triangle and go to opposite corner
        DecompressVertices(O[1]); }                         # starts vertices decompression

RECURSIVE PROCEDURE DecompressConnectivity(c) {
        REPEAT {                                            # Loop builds triangle tree and zips it up
                T++;                                        # new triangle
                O[c] = 3T; O[3T] = c;                       # attach new triangle, link opposite corners
                V[3T+1] = c.p.v; V[3T+2] = c.n.v;           # enter vertex Ids for shared vertices
                c = c.o.n;                                  # move corner to new triangle
                Switch READ(clers) {                        # select operation based on next symbol
                        Case 0: {O[c.n] = –1; V[3T] = ++N;} # C: left edge is free, store ref to new vertex
                        Case 110: { O[c.n] = –2; zip(c.n); } # L: orient free edge, try to zip once
                        Case 101: { O[c] = –2; c = c.n }    # R: orient free edge, go left
                        Case 100: { DecompressConnectivity(c);  c = c.n }   # S: recursion going right, then go left
                        Case 111: { O[c] = –2; O[c.n] = –2; zip(c.n);  RETURN }}}} # E: zip, try more, pop

RECURSIVE PROCEDURE Zip(c) {                                # tries to zip free edges opposite c
        b = c.n; WHILE b.o>=0 DO b=b.o.n;                   # search clockwise for free edge
        IF b.o != –1 THEN RETURN;                           # pop if no zip possible
        O[c]=b; O[b]=c;                                     # link opposite corners
        a = c.p;  V[a.p] = b.p.v;                           # assign co-incident corners
        WHILE a.o>=0 && b!=a DO {a=a.o.p; V[a.p]=b.p.v};
        c = c.p; WHILE c.o >= 0 && c!= b DO c = c.o.p;      # find corner of next free edge on right
        IF c.o == –2 THEN Zip(c) }                          # try to zip again

RECURSIVE PROCEDURE DecompressVertices(c) {
        REPEAT {                                            # start traversal for triangle tree
                U[c.t] = 1;                                 # mark the triangle as visited
                IF c.v.m != 1                               # test whether tip vertex was visited
                  THEN {G[++N] = c.p.v.g+c.n.v.g-c.o.v.g+READ(delta);   # update new vertex
                        M[c.v] = 1;                         # mark tip vertex as visited
                        c = c.r;}                           # continue with the right neighbor
                ELSE IF c.r.t.u == 1                        # test whether right triangle was visited
                    THEN IF c.l.t.u == 1                    # test whether left triangle was visited
                            THEN  RETURN                    # pop
                            ELSE  { c = c.l }               # move to left triangle
                      ELSE IF c.l.t.u == 1                  # test whether left triangle was visited
                            THEN { c = c.r }                # move to right triangle
                            ELSE { DecompressVertices (c.r);# recursive call to visit right branch first
                                   c = c.l } } }            # move to left triangle
```

Figure 3.6: The Edgebreaker decompression algorithm pseudocode

4. Documentation

The User manual and programmers documentation for the Morphome3cs project are stored in the project repository. We will talk about specific aspects of both these areas, which relate to the compression module of Morphome3cs project.

4.1 User manual

As the compression is used only to save and load triangle meshes, it is not very complicated to use. When we run the Morphome3cs program, we want to open a Mesh Viewer to make any use from the compression. We will open the mesh we want to compress. We want our mesh to have no holes or other defects to be able to save it in compressed format. Morphome3cs has a tool for our need: the Mesh Mending tool. We can open it by clicking on Tools and there by clicking on Mesh mending... . To do this, we need to have our mesh already loaded!

We want to set several options on the Mesh mending control panel. We want to check every single one of the following options:

- remove unconnected clusters
- fill holes
- remove unconnected vertices
- remove zero volume faces
- interpolate identic faces

When we have them checked, we choose a name for our repaired mesh and click the OK button. After a brief saving procedure, we need to re-open the repaired mesh. We can now click the save button and choose the name and the EBC format. Click save and the mesh is compressed and saved.

We don't need to do the repairing everytime we wish to save a mesh, but it is necessary to do it for the first time we work with that particular mesh to be able to use the EBC format or to be entirely sure that the mesh we wish to save satisfies all the needed criteria. It needs to be connected, 2-manifold equivalent mesh with no holes.

To load a mesh saved as EBC, we should click file and open. We choose the mesh we desire to open in the dialog and load it. The mesh will decompress and we are ready to work with it.

4.2 Technical documentation

The compression module for the Morphome3cs program is written exclusively in C#programming language. It is modular itself, so for example if we are content with conversion from mesh to corner table and uncomfortable with a compression scheme itself, we can leave conversion as it is and change the compression scheme only. Its the same with decompression or with creating the whole new compression method. To do this, we simply inherit interfaces $IMesh2CompressionFormat$ and $ICompressionFormat2Mesh$ to create conversion for the new compression and back or interfaces $ICompression$ and $IDecompression$ to create the compression and decompression methods themselves. The architecture of our solution is presented on the following figure. The interfaces are yellow, the classes are green and the huge blue boxes are the namespaces in the project. The left-to-right arrows are the saving and the right-to-left arrows are the loading part of the solution.

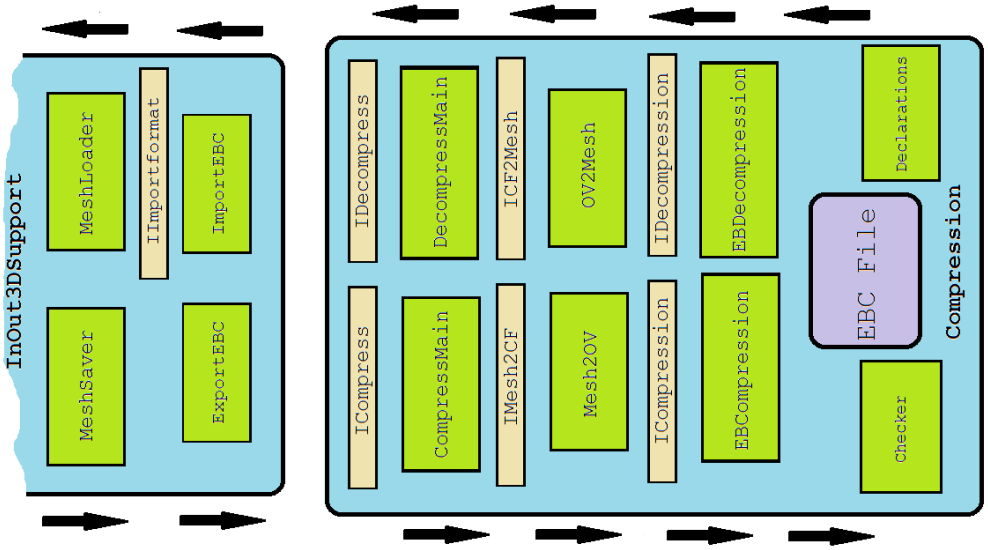


Figure 4.1: the architecture of the solution diagram

4.2.1 General process description

When a user clickes the save button, the Graphic User Interface (GUI) calls the MeshSaver class. It processes the input provided by the GUI and decides which saving method it will call. In our case, it calls the ExportEBC class and provides it with the mesh a user wants to save. The ExportEBC class transfers this mesh to the ICompress interface, which calls the IMesh2CompressionFormat (IMesh2CF on the diagram). This interface converts the input data into desired compression format and returns a data structure representing it. The data structure is defined in the Definitions class in the Compression modul and it is an abstract class. IMesh2CF receives Mesh and returns a data structure designed for the particular compression. The Icompress interface takes output of IMesh2CF and gives it to

the ICompression interface, which compresses it and returns a simple byte array. The ICompress converts it into a stream and returns this stream as its output.

When a user clickes the load button, the GUI calls the MeshLoader class. It processes the input and calls a method based upon the suffix of the input file. This method implements the IImportFormat interface, which specifies what methods must an imported have. We are intrested in .EBC, so the MeshLoader calls the ImportEBC class and provides it with the input file. The ImportEBC class calls the IDecompress interface and provides it with the input stream. This interface converts the stream into a byte array and calls the IDecompression interface with this byte array. IDecompression decodes the array, transforming it into its compression format and returns it. IDecompress gives obtained compression format to the ICompressionFormat2Mesh (ICF2Mesh on the diagram) interface, which converts the compression format back to Mesh data structure. The Mesh is the output of the load process, so IDecompress returns this Mesh to the MeshLoader.

If we would like to create a new compression scheme, we should use these interfaces and compression format data structure and inherit them, because they provide the basic compression and decompression logic. Our only problem would be only to implement particular classes and methods. Everything else is done, we are not needed to deal with the implementation into the Morphome3cs project, these interfaces already provide it.

4.2.2 Implementation details

The ICompress interface is implemented by the *CompressMain* class. It does exactly what is described above. It creates IMesh2CF and ICompression interfaces, takes the input Mesh, gives it to the IMesh2CF, takes result of this interface, gives it to ICompression interface and returns its result as a MemoryStream.

The *CompressionFormat* class is inherited by class called *Sestava*. It defines O and V arrays and vertices array for the corner table, T array as the index of textures for all the corners, the textures array, which contains all the texture coordinates, normals array, which contain the vertex normals and the Relation between the textures and vertices and the additional textures arrays which contain indexes to texture array and are used to compress textures with algorithm described above.

The *Mesh2OV* class implements the IMesh2CompressionFormat interface. It converts Mesh to the Sestava data structure, which it returns. The O and V table as well as the vertices and normals arrays are converted using the algorithms discussed before. The textures are preprocessed here to the structure needed by the algorithm for texture comrpession as well. It is all done within the convertMesh method, defined by the IMesh2CF interface.

The *EBCompression* class implements the ICompression interface. The in-

terface method comrpress takes the CompressionFormat data structure – it is the inherited Sestava class in this case – and creates the compressed byte array from it. This class implements the Edgebreaker algorithm directly, along with the prediction schemes for both the geometry of the triangle mesh and the normal coordinates for vertices, as well as the textures. It is all done by implementing algorithms discussed before. When the compression with these algorithms is done, the results are compressed by the arithmetic coder and put together into a one huge byte array, which is returned as a result.

The IDecompress interface is implemented by the class *DecompressMain*. This class acquires a stream of compressed data and converts it into a byte array which passes to the IDecompression interface. IDecompression returns CompressionFormat, DecompressMain gives it to the ICF2Mesh, which converts it to the Mesh data structure. This Mesh is returned as a result.

The *EBDecompression* class is the implementation of the IDecompression interface. It gets the compressed byte array, decomposes it into original parts, runs the arithmetic decoder on each part and runs the Edgebreaker decompression algorithm. Along with the connectivity decompression, it also decodes the texture marks and with the geometry decompression it decodes the normals and other vertex property. After the decompression procedure is done, this class converts the result into the Sestava data structure, which is returned as a result. This class also processes obtained textures in a way described before, to have the unique textures once again, since the compression may have created some duplicities.

The implementation of the ICF2Mesh interface, the *OV2Mesh* class gets the Sestava data structure from the IDecompress interface, runs the conversion from a corner table to a Mesh along with the conversion from the vertex normals to the facial normals. The Mesh data structure created in this class is returned as a result of loading procedure.

The *Declarations* class contains various data structures used in the compression scheme in general. The needs of the compression scheme are slightly different than the rest of the Morphome3cs project, the data structures needed for compression are defined in this class. The CompressionFormat class and the Sestava class are defined here as well.

The *Checker* class is the class, which implements the control algorithm to test the identity of two triangle meshes. It is not used in the compression scheme. This test was used for the testing of the correctness of the compression and decompression algorithm during the developement of this module. It can be used again, if anyone decides to develop another compression for Morphome3cs or if anyone decides to try and improve this compression module.

The compression module contains a few more classes not mentioned here, as they are of no importance to the Morphome3cs project or use of the compression tool. They were used only for debugging and are left where they are in case we will want to add anything new to the project.

5. Statistics and the results

5.0.3 Used specimens

We have six testing meshes at our desposal. They have quite boring names, which are $zena9 - 12$, $muz9$ and $muz10$. The testing data are the faces of anonymous poeple which were granted to me by other Morphome3cs developers. This data, along with the files compressed to compare the efficiency of the algorithm, are stored on the DVD attached to this paper. Don't open any other than the full versions of the specimens, or the obj. The other versions are created just for testing purposes. Lets talk about these specimens a little bit.

The $zena9$ is composed of 255747 vertices and vertex normals, 511538 faces and about a million texture coordinates. I don't know the exact number of textures, because it is not stored in the obj file which was given to me.

The $zena10$ is composed of 220757 vertices and vertex normals, 441558 faces and again, over a million of texture coordinates.

These two are the only specimens with the vertex normals I had in my disposal, but since the compression algorithm for normals is not complicated at all and it works for both of them, I consider it enough.

The $zena11$ has 184587 vertices. It has no vertex normals, as mentioned before. It has again, about a million texture coordinates and 369230 faces.

The $zena12$ specimen consists of 189827 vertices, has no normals, has the usual ammount of texture coordinates and 379682 faces.

The $muz9$ is composed of 211970 vertices, has no vertex normals, has around a million texture coordinates and 423968 faces.

The last specimen, the $muz10$, is my favourite. It look like a little boy and has 162639 vertices, no vertex normals, the usual ammount of texture coordinates and 325386 faces.

5.0.4 The statistics and tests

We test the algorithms efficiency by comparing it to the original size of the obj file format. We will compress various parts of the meshes and observe the ammount of space the result needs to be stored.

We start with the full compression scheme. As we compare the average ammount of space needed to store the obj file (the average is 33,3 MB) with the

average compressed data (4,3 MB), we can clearly see that our algorithm is prette decent, with its achieved compress ration being 13% (the obj file taken as the original data ammount) ! It is quite an achievement, because the lossless comrpession is considered succesfull if it has a compress ratio around 50% in general.

We will now compare the effectivenes of our implementation of the Edgebreaker algorithm with the original version presented by prof. Jarek Rossignac. We keep in mind, that our algorithm is lossless and used for compression with higher precision than the original version. We use the two remaining files for each specimen, one is suffixed base and the other is suffixed clers. The base files contain the clers string, the vertex data and the handles. The clers files contain only the clers strings, to be able to measure the effectivness of the compression of the connectivity.

The original Edgebreaker algorithm used only 12 bits per vertex to compress a geometry of the triangle mesh. Our algorithm needs 1698816 Bytes to compress a single mesh in average. It is approximately 66 bits per vertex.

The connectivity compression was slightly less then one bit per triangle. Our compression for one triangle is around 1,2 bits per triangle, which can be easily counted From the files.

Because we save the data using less space than was needed for the obj file, we must pay the prize somewhere else. That prize is the saving and the loading time, which is approximately five times higher than for the obj.

Conclusion

I would like to sum up on my work. I was able to create a functional implementation of the Edgebreaker algorithm and expand the algorithm with methods that compress both, the vertex normals and the texture coordinates with satisfying efficiency for a lossless compression scheme. It is approximately eight times more efficient than the often used obj file format.

This solution is created as a part of already functional project Morphome3cs, developed on Faculty of Mathematics and Physics in cooperation with Department of Anthropology and Human Genetics on the Faculty of Science at Charles University in Prague. The compression module is created to save a noticable ammount of space and to provide the interfaces for the possible future designs. To write a code in such a large project was a good experince for me, as it wasn't easy. I think I learned a lot by creating this program about coordination in large projects.

The compression could be improved a bit more. The entropy coder used in this solution works really well, but it uses a model for general purposes. The room for improvement is at hand. If we write a model specifically for this compression, we could improve the compression effectivenes. We could think about different and maybe more effective schemes of compression for the individual vertex or facial attributes as well. We could create a setting bar for the compression, so the user can choose which attributes he wishes to compress. Or we can use the interfaces to create a totally different compression scheme.

The bibliography

[1] Rossignac Jarek. *Edgebreaker: Connectivity compression for triangle meshes*. GVU Center, Georgia Institute of Technology GVU Technical Report GIT-GVU-98-35 (revised version of GIT-GVU-98-17).

[2] Rossignac Jarek and Szymczak Andrzej. *Wrap&Zip decompression of the connectivity of triangle meshes compressed with Edgebreaker*. GVU Center, College of Computing, Georgia Institute of Technology, Atlanta GA 30332-0280.

[3] Rossignac, Jarek. *3D Compression Made Simple: Edgebreaker on a Corner-Table*. College of Computing and GVU Center Georgia Institute of Technology

[4] Rossignac Jarek, Safonova Alla and Szymczak Andrzej. *Edgebreaker: Connectivity compression for triangle meshes*.

[5] Žára Jiří, Beneš Bedřich, Sochor Jiří and Felkel Petr. *Moderní počítačová grafika*. druhé vydání Computer press, nám 28. dubna Brno ISBN 80-251-0454-0

[6] *Apache Subversion [online]*. http://subversionàpacheòrg/

[7] Masařík Tomáš. *Mesh mending. Praha, 2011. Bakalářská práce.* Univerzita Karlova, Matematicko-fyzikální fakulta, Kabinet software a výuky informatiky.

[8] *Wavefront òbj File [online]*. Dostupné z: *http* : *//en.wikipedia.org/wiki/Wavefront_obj_file*

[9] *Morphometrics [online]*. http://cggmffčuničz/trac/morpho/

[10] *Morphome3cs: User Manual [online]*. http://cggmffčuničz/trac/morpho/attachment/wiki/

[11] *Morphome3cs 2: Developer Manual [online]*. http://cggmffčuničz/trac/morpho/attachment/wiki/WikiStart/3-developer-manualṗdf

Attachments

DVD content

- Morphome3cs project
- The set of testing data

Used pictures